High-Performance Graphics 2023
J. Bikker and C. Gribble
(Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 8

Edge-Friend: Fast and Deterministic Catmull-Clark Subdivision Surfaces

Bastian Kuth[1] Max Oberberger[2] Matthäus Chajdas[2] Quirin Meyer[1]
[1]Coburg University of Applied Sciences and Arts, Germany
[2]AMD, Germany

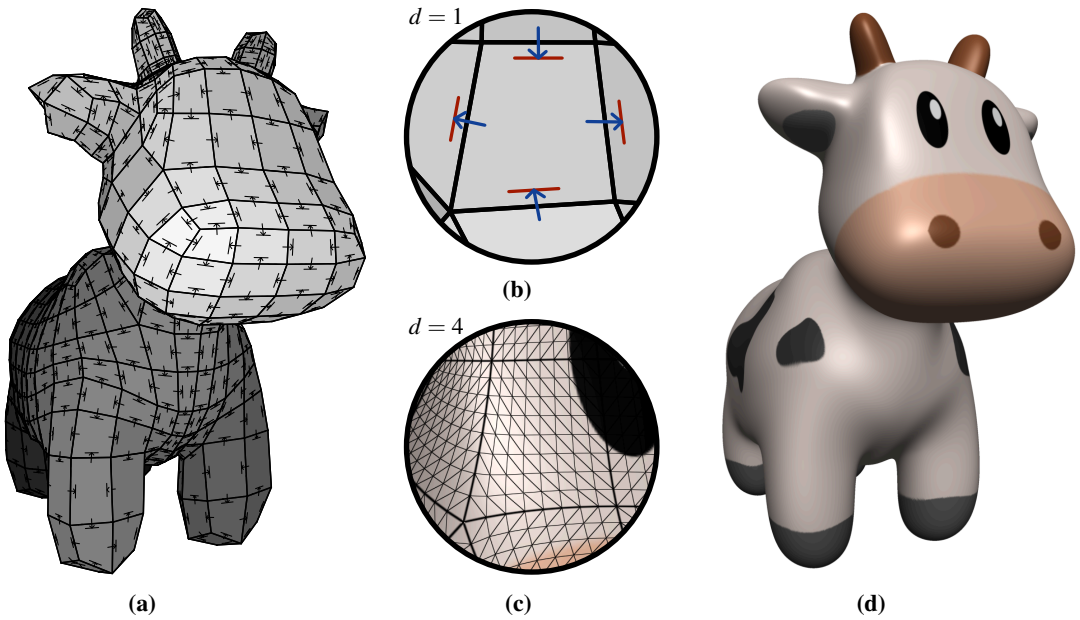

Figure 1: *(a) A control mesh after a pre-processing subdivision iteration is quad-only. (b) Our edge-friend data structure implicitly assigns two opposing edges (red) to each quad. Each quad stores only two edges in neighboring quads (blue). (c) We refine the edge-friend structure breadth-first down to level $d = 4$. (d) Refining and rendering the shown model takes under 40µs on an AMD Radeon RX 7900 XTX GPU.*

Abstract

We present edge-friend*, a data structure for quad meshes with access to neighborhood information required for Catmull-Clark subdivision surface refinement. Edge-friend enables efficient real-time subdivision surface rendering. In particular, the resulting algorithm is deterministic, does not require hardware support for atomic floating-point arithmetic, and is optimized for efficient rendering on GPUs. Edge-friend exploits that after one subdivision step, two edges can be uniquely and implicitly assigned to each quad. Additionally, edge-friend is a compact data structure, adding little overhead. Our algorithm is simple to implement in a single compute shader kernel, and requires minimal synchronization which makes it particularly suited for asynchronous execution. We easily extend our kernel to support relevant Catmull-Clark subdivision surface features, including semi-smooth creases, boundaries, animation and attribute interpolation. In case of topology changes, our data structure requires little pre-processing, making it amendable for a variety of applications, including real-time editing and animations. Our method can process and render billions of triangles per second on modern GPUs. For a sample mesh, our algorithm generates and renders 2.9 million triangles in 0.58ms on an AMD Radeon RX 7900 XTX GPU.*

CCS Concepts

• ***Computing methodologies*** → ***Rendering; Parametric curve and surface models; Massively parallel algorithms;***

1. Introduction

Catmull-Clark subdivision [CC78] is a surface modeling algorithm that generates a dense and smooth quad mesh from a sparse polygon *control mesh*. Today, it is a widespread modeling technique. A *subdivision step* splits the control-mesh polygons into quads and transforms the quads' positions according to refinement rules. The resulting mesh is continuously subdivided until it is dense enough.

The use of graphics processing units (GPUs) proved to be advantageous: GPU data-parallelism assists in achieving high subdivision speed. Moreover, the final mesh can be rendered directly from GPU memory. However, parallelizing subdivision is not trivial and existing approaches suffer from draw-backs which we improve on with the following contributions:

A novel quad-mesh-connectivity data structure. Subdivision requires neighbor information through suitable data structures. Existing ones suffer from a large memory foot-print [PEO09, MWS*20, DV21]. We propose a more *light-weight data structure* that allows to quickly access neighbor information. We show that it is sufficient to store only two references to neighbor edges per quad, called *edge-friends*, see Fig. 1. Since subdivision tends to be a memory-bound problem, this increases subdivision speed. Furthermore, we organize edge-friend in a *spatially coherent memory layout*, which makes memory access patterns beneficial for GPU performance.

An atomic-operation free gathering approach. Implementations using atomic floating-point operations [PEO09, MWS*20, DV21] come with a performance penalty and require vendor- and API-specific extensions. Moreover, the non-deterministic scheduling of atomic operations changes their order on a frame-by-frame basis. With floating-point arithmetic not being associative, flickering artifacts can occur, even between frames with identical inputs. We completely *eliminate atomic operations* by expressing subdivision with *gather operations* as opposed to atomic scatter operations.

A single synchronization barrier. Many existing GPU approaches [PEO09, MWS*20, DV21] need multiple dependent compute-kernel dispatches with expensive barrier synchronizations for a single subdivision step. We require only *a single dispatch*, and thus *a single synchronization barrier* per subdivision iteration, which provides additional performance benefits.

Low pre-processing cost. Some methods trade fast surface evaluation against expensive pre-processing [NLMD12, Pix22]. This slows down modeling, animation, and simulation tasks that require topology changes like face, edge, or point insertion and deletion. Our approach requires *negligible pre-processing* enabling real-time geometric and topological edits.

Simple and extensible. Additionally, our subdivision compute kernel is simple and we demonstrate how to easily integrate relevant subdivision features like boundaries [Nas87], semi-smooth creases [DKT98], animation, and attribute interpolation, which makes our method attractive for production environments.

However, our approach possesses the following **limitations**: Surface evaluation with hardware tessellation [NLMD12, Pix22] remains faster, but requires substantially more pre-processing than our method. We obtain conformal, crack-free meshes with uniform subdivision; however, like other methods [MWS*20, DV21], we do not handle crack-free adaptive subdivision.

2. Related Work

Catmull-Clark Subdivision Rules [CC78] update the input-mesh vertices and create one new point per face and edge. A *new face-point* is the centroid of the face. A *new edge-point* is the average of the edge's incident new face points f_0, f_1 and vertices v_0, v_1:

$$e = \frac{1}{4}(v_0 + v_1 + f_0 + f_1). \tag{1}$$

With the valence n of an old vertex v, the average Q of all its adjacent face points, and the average R of the midpoints of all edges incident to v, we get the *new vertex point*

$$v' = \frac{1}{n}(Q + 2R + (n-3)v). \tag{2}$$

An *extra-ordinary vertex* has valence $n \neq 4$. One vertex- and face-point, as well as two edge-points define a new quad.

Patch-based Subdivision methods split the control mesh into patches. A patch consists of a face together with the context required for local subdivision [BS02]. While leading to data duplication and redundant computations, each patch can be treated independently enabling both parallel and adaptive subdivision. Extra triangles close cracks between different levels. However, floating-point rounding errors cause cracks, but a correct operation order assures consistent results where patches meet [NLMD12].

Breadth-First Subdivision algorithms apply the subdivision rules on an entire mesh in parallel. Our method falls into this category. Methods build mesh data structures that contain neighbor information [DV21, PEO09, MWS*20]. For example, to compute an edge-point, the data structure must contain information about faces connecting to an edge. Patney et al. [PEO09] represent a mesh with a vertex-, a quad-, and an edge-buffer, where one edge references two quads and two vertices. They achieve view-adaptive, crack-free tessellation. Mlakar et al. [MWS*20] describe topology with sparse matrices. Combined with specialized linear algebra kernels, they achieve fast breadth-first subdivision, but require a complex implementation. Dupuy and Vanhoey [DV21] achieve similar performance. They represent a mesh using the established halfedge data structure and convert the original subdivision rules to work on a per-halfedge basis.

Direct Evaluation Catmull-Clark subdivision surfaces generalize uniform bi-cubic tensor-product B-Spline surfaces and overcome their topological limitations. Both surfaces are identical for quads with eight adjacent quads. There, direct B-Spline evaluation is more efficient than subdivision. As the refinement rules maintain the number of extra-ordinary vertices, the directly evaluable proportion of the surface grows with each subdivision. Specialized methods exist for direct evaluation of quads with one isolated extra-ordinary vertex [Sta98], at an extra-ordinary boundary [LB07], and with a single semi-sharp crease [NLG12]. For other configurations, we are only aware of approximations [LS08]. Hybrid methods [NLMD12, BFK*16] subdivide until direct evaluation is possible. These methods leverage hardware tessellation minimizing I/O and enabling adaptive rendering.

3. Edge-Friend

We first describe the *creation* of our edge-friend data structure. Next, we derive *edge-friend refinement rules* and present a cache-coherent *vertex-memory layout*. Furthermore, we demonstrate how edge-friend handles important *extensions*, including boundaries. Finally, we discuss the *rendering and attribute interpolation* of our subdivided mesh.

3.1. Creation

We obtain an edge-friend mesh during a first subdivision iteration. This can be performed as pre-process or every frame with any subdivision method supporting arbitrary polygons. After one subdivision, we obtain a quad-only mesh necessary for our data structure.

Let the index buffer of the d-th subdivision be $\mathcal{I}_d$ (cf. Fig. 2b), where level $d = 0$ is the unprocessed input. We call an element of the index buffer a *corner* [RSS03], see red numbers in Fig. 2b. Each input corner of $\mathcal{I}_{d-1}$ maps to a new quad in $\mathcal{I}_d$. Here, we make our key observation: the corners inside a quad can be rotated arbitrarily without causing topological changes. We always start a new quad with the vertex index of the old corner, continue with the adjacent edge-point and face-point along the winding order and conclude with the second edge-point. Given a quad with the corners (v_0, v_1, v_2, v_3), we call the opposing edges $\overline{v_0v_1}$ and $\overline{v_2v_3}$ the *on-edges* of the quad. The opposing edges $\overline{v_1v_2}$ and $\overline{v_3v_0}$ are called the *off-edges* of the quad. Fig. 2a shows an example with on-edges marked by additional lines. Fig. 2b shows the converted index buffer $\mathcal{I}_d$ and its corners c.

For the new index buffer, the following properties hold:

- $(4c+0, 4c+1, 4c+2, 4c+3)$ addresses a quad.
- $(2c+0, 2c+1)$ addresses an edge.
- (c) addresses a corner.

An on-edge of one quad is an off-edge in an adjacent quad. In Fig. 2, $\overline{11\ 7}$ is an on-edge of quad 0, but an off-edge in quad 2.

Let c be a corner index, e be an edge index and $\oplus$ be a bit-wise exclusive or operation. Using bit-logic

- $\textsc{Quad}(c) = \left\lfloor \frac{c}{4} \right\rfloor$ yields the quad index of a corner,
- $\textsc{Equad}(e) = \left\lfloor \frac{e}{2} \right\rfloor$ yields the quad index of an edge,
- $\textsc{Diag}(c) \quad = c \oplus 2$ yields the corner index across the quad, and
- $\textsc{Off}(c) \quad = c \oplus 3$ yields the corner index along the off-edge.

To add neighborhood information, we assign both off-edges of a quad the corresponding on-edge indices. Thus, each quad consists of two opposing on-edges and references two on-edges of neighboring quads. We call these references the *edge-friends* of a quad. We store edge-friends in a *friend buffer* $\mathcal{G}_d$, where d is the subdivision level. An element of $\mathcal{G}_d$ is the tuple (g_0, g_1), where g_0 is the friend of off-edge $\overline{v_1v_2}$ and g_1 the friend of off-edge $\overline{v_3v_0}$. Fig. 2b shows an example of $\mathcal{G}_d$.

For each vertex, we select an arbitrary corner and make that corner index a vertex attribute. We call this attribute *valence loop start*, as it enables gathering points from adjacent faces to update a vertex position. This allows atomic-operation free vertex updates. We denote the respective buffer $\mathcal{L}_d$, exemplified in Fig. 2c.

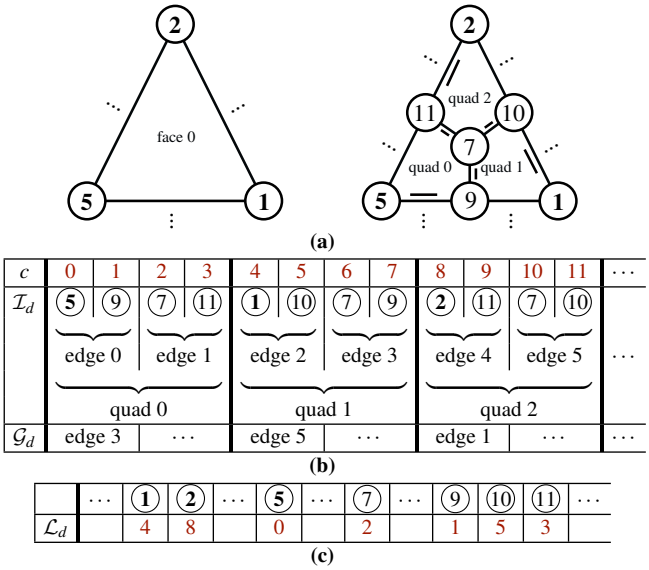

Figure 2: *Edge-friend Data Structure. (a) During pre-processing, each corner of the original mesh (left) maps to a quad in the subdivided mesh (right). We mark the on-edges of each quad with extra edge lines. Vertices are shown as circled numbers and original mesh vertices are shown in bold. (b) Using corner indices c (red numbers), we obtain quads and edges from the index buffer $\mathcal{I}_d$. The edge-friend buffer $\mathcal{G}_d$ is used to access neighborhood information. (c) To gather neighboring vertices, $\mathcal{L}_d$ maps each vertex to a corner. This corner must in turn reference the vertex.*

We get the buffer sizes from the vertex and face count V_d and F_d:

$$V_d = |\mathcal{V}_d| = |\mathcal{L}_d|, F_d = \frac{1}{4}|\mathcal{I}_d| = |\mathcal{G}_d|.$$

The buffer sizes increase exponentially with every subdivision step:

$$V_{d+1} = V_d + 3F_d, \quad F_{d+1} = 4F_d.$$

3.2. Edge-friend Refinement

We split edge-friend subdivision into two tasks: the *quad task* and the *vertex task*. Both tasks run in the same compute shader kernel.

Quad Task With one thread per quad, the quad task computes the face-point, the two edge points of the off-edges of the quad, the indices for four new quads, and eight new edge-friends. First, we load both edge-friends $(g_0, g_1) \leftarrow \mathcal{G}_d[q]$ of the current quad with index q. Remember that g_i are edge indices. Hence, we obtain the corresponding corner indices by $2g_i + 0$ and $2g_i + 1$. Then, we can read all required vertex indices, as shown in Fig. 3a:

$$\begin{aligned} v_0 &= \mathcal{I}_d[2g_1 + 0], & v_0' &= \mathcal{I}_d[\textsc{Off}(2g_1 + 0)], \\ v_1 &= \mathcal{I}_d[2g_0 + 1], & v_1' &= \mathcal{I}_d[\textsc{Off}(2g_0 + 1)], \\ v_2 &= \mathcal{I}_d[2g_0 + 0], & v_2' &= \mathcal{I}_d[\textsc{Off}(2g_0 + 0)], \\ v_3 &= \mathcal{I}_d[2g_1 + 1], & v_3' &= \mathcal{I}_d[\textsc{Off}(2g_1 + 1)]. \end{aligned}$$

With v_i, v_i', we load the vertex positions, compute the face-point

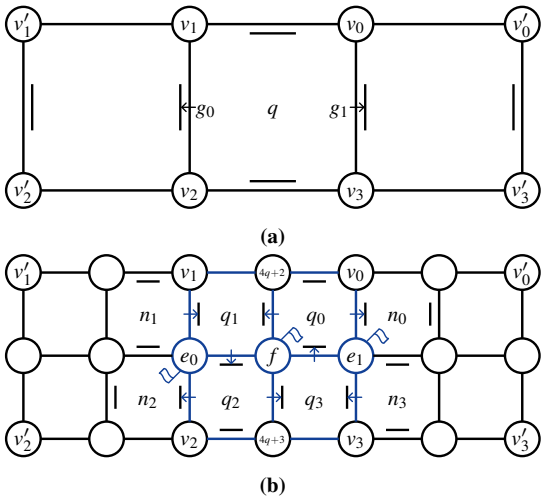

Figure 3: *Quad Task. (a) Each quad task uses the edge-friends g_i of quad q to load the vertices v_i and v'_i. (b) Next, the quad task computes the new face-point of q and stores it at index f in $\mathcal{V}_{d+1}$. Additionally, it computes the new edge-points and stores them at e_i in $\mathcal{V}_{d+1}$. With v_i, e_i, f, $4q+2$, and $4q+3$ the new quads q_i are written to $\mathcal{I}_{d+1}$. For the friend relations, shown as arrows, we require the neighbor quads n_i. Finally, the quad task writes the three new valence loop start corners for the generated vertices to $\mathcal{L}_{d+1}$. All elements generated by the quad task are highlighted in blue.*

of the centering quad and the edge points of the off-edges according to Eq. (1). We store the generated face-point in the new vertex buffer at index $f = 4q+1$, and the generated edge-points at

$$e_0 = 4(\textsc{Equad}(g_0)+2+(g_0 \bmod 2)),$$
$$e_1 = 4(\textsc{Equad}(g_1)+2+(g_1 \bmod 2)),$$

i.e., next to the face-point of the friends of q. This improves caching, as shown in Sec. 3.3. Moreover, the quad task writes the four new quads emerging from the old quad. The locations in the new index buffer for the four new quad indices $q_i, i \in [0,3]$ are $q_i = 4q+i$. Again, the new quads start at an original vertex position, continue with an edge-point, the face-point and conclude with the second edge-point:

$$\mathcal{I}_{d+1}[4q_0 : 4q_1] \leftarrow (w_0, 4q+2, f, e_1),$$
$$\mathcal{I}_{d+1}[4q_1 : 4q_2] \leftarrow (w_1, e_0, f, 4q+2),$$
$$\mathcal{I}_{d+1}[4q_2 : 4q_3] \leftarrow (w_2, 4q+3, f, e_0),$$
$$\mathcal{I}_{d+1}[4q_3 : 4q_3+4] \leftarrow (w_3, e_1, f, 4q+3),$$

where w_i is the index of v_i in the new vertex buffer. To add the new friend relations, we need new neighboring quads indices

$$n_0 = 4\textsc{Equad}(g_1)+2(g_1 \bmod 2)+0,$$
$$n_1 = 4\textsc{Equad}(g_0)+2(g_0 \bmod 2)+1,$$
$$n_2 = 4\textsc{Equad}(g_0)+2(g_0 \bmod 2)+0,$$
$$n_3 = 4\textsc{Equad}(g_1)+2(g_1 \bmod 2)+1,$$

where n_i is adjacent to q_i. Using this, we can compute and write eight new friend indices:

$$\mathcal{G}_{d+1}[q_0]_0 \leftarrow 2q_1+1, \quad \mathcal{G}_{d+1}[q_0]_1 \leftarrow 2n_0+0,$$
$$\mathcal{G}_{d+1}[q_1]_0 \leftarrow 2q_2+1, \quad \mathcal{G}_{d+1}[n_1]_1 \leftarrow 2q_1+0,$$
$$\mathcal{G}_{d+1}[q_2]_0 \leftarrow 2q_3+1, \quad \mathcal{G}_{d+1}[q_2]_1 \leftarrow 2n_2+0,$$
$$\mathcal{G}_{d+1}[q_3]_0 \leftarrow 2q_0+1, \quad \mathcal{G}_{d+1}[n_3]_1 \leftarrow 2q_3+0.$$

To conclude the quad task, we write the new valence loop start corners for the three newly generated vertices. It is valid to choose any of the connected corners, and we choose

$$\mathcal{L}_{d+1}[f] \leftarrow 4q_0+2,$$
$$\mathcal{L}_{d+1}[e_0] \leftarrow 4n_2+1,$$
$$\mathcal{L}_{d+1}[e_1] \leftarrow 4n_0+1.$$

For an example, see Fig. 3b.

Vertex Task The vertex task updates each vertex position. In order for our algorithm to work in a single dispatch, we need to reformulate the vertex-point update rule to not depend on any points computed during the same subdivision step. After our required preprocessing iteration, the mesh is quad-only. According to the supplemental material of de Goes et al. [dGDMD16], given the valence n of vertex-point v_d, the n points connected to v_d with an edge E and the n points diagonally connected to v_d with a quad F, the new vertex-point v_{d+1} is

$$v_{d+1} = \alpha v_d + \frac{\beta}{n}\sum_j^n E_j + \frac{\gamma}{n}\sum_j^n F_j, \tag{3}$$

where $\alpha = 1-\beta-\gamma$, $\beta = \frac{3}{2n}$, and $\gamma = \frac{1}{4n}$. We collect the two sums by iterating over the faces adjacent to the input vertex index v, as shown in Algorithm 1. The loop starts at the corner $c \leftarrow \mathcal{L}_d[v]$. We compute the required point locations using the previously established bit logic on corners, and add the points to their respective accumulators. To visit the next corner, one of the two friend references of the current quad is used, depending on which off-edge the current corner lies. We loop around the vertex until we arrive at the starting corner. We then compute the new vertex point position with Eq. (3). Typically, the new vertex index is $w = 4v$, the other case is discussed in Sec. 3.3. An old corner c maps to a new corner c' with $c' = 4c$. We can thus propagate the input valence loop start to $\mathcal{L}_{d+1}$.

Task Merge For closed topology of genus 0 with the total number of vertices V, the number of edges E and the number of faces F, the Euler characteristic states: $V-E+F=2$. As we have uniquely assigned two edges to each quad, it applies that $E = 2F$. Thus, the number of quad and vertex tasks per closed topology is almost equal: $V = F+2$. In addition, we do not require one task to finish before the other and thus can merge both tasks into a single compute shader. For the exceeding vertices, the kernel can just terminate early. This simplifies the implementation and is faster to run, as the algorithm only requires a single synchronization barrier between each subdivision iteration.

3.3. Vertex Memory Layout

Methods such as the halfedge refinement by Dupuy and Vanhoey [DV21] store the new face- and edge-points behind the

Algorithm 1 Vertex Task

procedure VERTEXTASK(vertex index v, old mesh $F_d, \mathcal{V}_d, \mathcal{I}_d, \mathcal{G}_d, \mathcal{L}_d$, new mesh $\mathcal{V}_{d+1}, \mathcal{L}_{d+1}$)
  $c \leftarrow \mathcal{L}_d[v]$
  $c' \leftarrow c$
  $e \leftarrow 0, f \leftarrow 0, n \leftarrow 0$
  repeat
    $n \leftarrow n+1$
    $e \leftarrow e + \mathcal{V}_d[\mathcal{I}_d[\text{OFF}(c')]]$
    $f \leftarrow f + \mathcal{V}_d[\mathcal{I}_d[\text{DIAG}(c')]]$
    $g \leftarrow \mathcal{G}_d[\text{CQUAD}(c')]$
    $i \leftarrow \begin{cases} 0, & \text{if } (c' \bmod 4) = 1 \vee (c' \bmod 4) = 2 \\ 1, & \text{otherwise} \end{cases}$
    $c' \leftarrow 2g_i + (c' \bmod 2)$
  until $c' = c$
  $w \leftarrow \begin{cases} 4v, & \text{if } v \le F_d \\ 3F_d + v, & \text{otherwise} \end{cases}$
  $\beta \leftarrow \frac{3}{2n}, \gamma \leftarrow \frac{1}{4n}, \alpha \leftarrow 1 - \beta - \gamma$
  $\mathcal{V}_{d+1}[w] \leftarrow \alpha \mathcal{V}_d[v] + \frac{\beta}{n} e + \frac{\gamma}{n} f$
  $\mathcal{L}_{d+1}[w] \leftarrow 4v$
end procedure

last memory address for the updated vertex positions. As a new quad always uses one vertex-point, one face-point, and two edge-points, the distance between the memory addresses accessed at once increases exponentially with every subdivision step. With our method, we can interleave the memory locations of the different point types to achieve better data locality. This speeds up both the next subdivision iteration and the final drawing of the generated geometry. Our vertex buffer is split into chunks of four positions. The first slot of a chunk i is reserved for the updated vertex-point position w_i of vertex i. The second slot is reserved for the new face-point f_i of quad i. The third and fourth slots are reserved for the edge-points e_{2i+0} and e_{2i+1} of the on-edges of quad i. As previously established, the number of vertices is unequal to the number of faces. Therefore, we have to compensate if i is greater to the number of faces F_d. An old vertex at index v maps to a new index w:

$$w \leftarrow \begin{cases} 4v, & \text{if } v \le F_d \\ 3F_d + v, & \text{otherwise.} \end{cases}$$

The resulting vertex buffer for a mesh with a single closed topology, where $V = F + 2$, looks like this:

$$\left[w_0, f_0, e_0, e_1, w_1, f_1, e_2, e_3, \cdots, e_{2F_d-2}, e_{2F_d-1}, w_{V_d-2}, w_{V_d-1}\right].$$

For other topological genera, the handling of exceeding buffer sizes works analogously.

3.4. Semi-Sharp Creases

A common extension of the Catmull-Clark subdivision rules is the use of semi-sharp creases [DKT98]. Individual edges of the control mesh can be assigned a sharpness value $\sigma \in \mathbb{R}_{\ge 0}$. The sharpness denotes whether to apply the original "smooth" rules from Sec. 2 ($\sigma = 0$), or to use additional "sharp", "crease" or "corner" rules in the current subdivision step ($\sigma \ge 1$). $0 < \sigma < 1$ denotes a blend between the rules. When a creased edge is subdivided, the two resulting edges receive the sharpness of the original edge minus one, thus $\sigma' = \text{MAX}(0, \sigma - 1)$. Face-points remain the same as is Sec. 2.

Edge-point Given the two points p_0 and p_1 that connect to an edge, the *sharp rule* is:

$$e_{\text{sharp}} = \frac{1}{2}(p_0 + p_1).$$

Given the smooth point e_{smooth} like in Eq. (1), and the sharpness σ of the edge, the resulting edge-point is:

$$e = \text{LERP}(e_{\text{smooth}}, e_{\text{sharp}}, \text{MIN}(\sigma, 1)). \tag{4}$$

Vertex-point Given two points p_0 and p_1 connected to a vertex v with two edges that have a sharpness $\sigma > 0$, the *crease rule* is

$$v_{\text{crease}} = \frac{1}{8}(p_0 + 6v + p_1),$$

and the *corner rule* is

$$v_{\text{corner}} = v.$$

Given the smooth point v_{smooth} like in Eq. (2), the number of edges m that are connected to vertex v with $\sigma > 0$, and the average sharpness $\overline{\sigma}$ of all these edges, the resulting updated vertex-point v' is:

$$v' = \begin{cases} v_{\text{smooth}}, & \text{if } m < 2 \\ \text{LERP}(v_{\text{smooth}}, v_{\text{corner}}, \text{MIN}(\overline{\sigma}, 1)), & \text{if } m > 2 \\ \text{LERP}(v_{\text{smooth}}, v_{\text{crease}}, \text{MIN}(\overline{\sigma}, 1)), & \text{otherwise.} \end{cases}$$

We assign a sharpness value to each edge-friend reference. The required adjustments to the edge point computation directly follows Eq. (4). In addition, we write out the new sharpness values of the generated friend relations. For the vertex-task, we add two more accumulators to the loop: m for the number of adjacent edges with sharpness $\sigma > 0$ and $\hat{\sigma}$ for accumulating all sharpness values. Thus $\overline{\sigma} = \frac{\hat{\sigma}}{m}$. For the crease rule, we require the two points p_0 and p_1. If the loop encounters a creased edge with a connected vertex e, we set $p_0 \leftarrow e$ if $m = 0$ and $p_1 \leftarrow e$ otherwise.

Some assets require smooth subdivision of the sharpness values, which is known as the *Chaikin rule* [Cha74]. If this is desired, one could move the refinement of the sharpness values to the vertex-task, because there we have access to neighboring sharpness values.

3.5. Meshes with Boundaries

Although our data structure relies on a closed mesh topology, we support mesh boundaries. As semi-sharp creases are supported, we can simply close the geometry with *ghost quads* and mark previous boundary edges as infinitely sharp. Each boundary consisting of k edges receives $\frac{k}{2}$ ghost quads arranged in a fan and one additional vertex in the center of this fan. With a subdivision iteration on preprocessing, where each edge gets split into two, it is given that $\frac{k}{2}$ is an integer. After subdivision, the ghost quads can simply be ignored for rendering. Some assets require boundary corners to follow the corner rule. To achieve this, we mark the ghost edges attached to this vertex as infinitely sharp.

3.6. Rendering

After subdividing the desired amount of iterations, we employ a mesh shader to render the refined meshes to the screen. Here, we have to handle the problem of attribute interpolation, most commonly of texture coordinates, but our approach can also be used for other face-varying attributes. While subdivision requires a closed mesh, texture mapping has to slice open a mesh to be able to project it onto the texture. Usually this can be handled by duplicating the vertices at texture map seams on asset creation. One vertex receives the texture coordinate of one side of a texture map seam, and the other one the coordinate of the other side.

To do this duplication in real-time, we form a meshlet from each set of subdivided quads that originate from a single quad of our pre-processed control mesh. This comes with the benefit that the duplication of the vertices along the edges where texture seams can happen is done implicitly and without requiring any additional memory. On pre-processing, we associate one quad with four texture coordinates, thus one per corner, which are loaded and linearly interpolated by the mesh shader. Note that this only covers one possible method for interpolation. OpenSubdiv provides other interpolation methods for face-varying attributes, where the attribute itself is subject to smooth subdivision [Pix22]. If the number of iterations is too great for the output triangle limit of a mesh shader, we additionally employ an amplification shader. The amplification shader splits the data of an original quad into tiles that are within the bounds of the triangle output limit.

The *surface normal vector* of a vertex is usually not defined by attributes, but by finding the partial derivatives of the refined surface. We compute the normal vectors using the usual cross-product formula in the mesh shader. If desired, it is possible to compute the limit surface normal by using the equations of Halstead et al. [HKD93]. This limit surface projection can also be applied to the vertex positions. The computation of the normal vectors and texture coordinates also allows for normal- and displacement mapping. The real-time interpolation of *vertex-blend attributes* for animation is not necessary, because the transformation of the positions is performed before real-time subdivision.

4. Results and Discussion

We evaluate the performance of our method on the meshes of Fig. 4. The collection includes regular meshes, meshes with boundaries, and with semi-sharp creases. Big Guy and Pig start at $d = 0$ as their initial control meshes already comply with our on-off-edge rule after rotating every second quad.

Consider the dual graph of the quad mesh. For our on-off-edge rule to function, we require a closed quad mesh where all dual chord rings [DSSC08] have a length of $2n$. This is true, if the closed quad mesh is homeomorphic to a sphere. In other cases, the pre-processing iteration doubles the length of all dual chord rings, forcing all dual chord rings to be of length $2n$. This assures that we always have a consistent edge-friend data structure.

For non-quad-only meshes we perform a pre-processing iteration on the CPU when loading the model. Remember that any subdivision method that supports arbitrary face sizes can be applied for this step. Tab. 1 provides timings of our naïvely parallelized implementation of the pre-process iteration. We use GPU timers to isolate performance measurements for both subdivision and rendering. All measurements were taken on an AMD Ryzen 9 5950X, together with an AMD Radeon RX 7900 XTX and an NVIDIA RTX 4080.

Meshes	Ogre	Big Guy	Pig	Spot	Rook	Bishop	Car	Imrod
Time (ms)	1.28	0.99	0.26	0.42	0.78	0.82	1.81	2.93

Table 1: *Pre-processing Time. Each cell is the pre-processing time in milliseconds required to create the edge-friend data structure for the meshes from Fig. 4 on the CPU. The measurements include the creation of a hash-map for mesh connectivity.*

We compare our method to the closely related halfedge refinement by Dupuy and Vanhoey [DV21]. We refer to it as *Halfedge*. For fair comparison, we ported over the publicly available OpenGL Halfedge implementation into our Direct3D12 application. As the input meshes are quad-only, we always employ the quad-only optimization of Halfedge, where the *Prev*, *Next*, and *Face* references of each Halfedge can be trivially computed. For simplicity reasons, we always use an implementation that supports semi-sharp creases for both Halfedge and our method, even if a test mesh does not have any edges tagged as such. Furthermore, as we use the regular crease refinement in our implementation, we removed the additional compute dispatch from Halfedge for refining the sharpness values according to the Chaikin rule. Atomic floating point arithmetic is not supported by all vendors or APIs. To take this case into account, we simulate atomic float addition similar to Patney et al. [PEO09].

Fig. 5 shows our subdivision benchmark results without rendering. As can be seen, our method outperforms Halfedge by a factor of about three. This does not change for meshes with boundaries, where our method has to additionally subdivide ghost quads. Furthermore, the figure reveals that simulating atomic float addition for Halfedge is evidently slower.

Fig. 6 shows the overall run-time required to subdivide and render a mesh. We achieve frame-times well above the required threshold for real-time rendering, even when generating more triangles than framebuffer pixels.

Both Halfedge and our algorithm require two temporary blocks of memory for subdivision, one for input and one for output. The semantics of both blocks are swapped each iteration: the old output becomes the new input and vice-versa. Based on the number of quads F and the number of vertices V, the required temporary memory sizes to hold a single iteration for Halfedge $h(F,V)$ and our algorithm $e(F,V)$ are

$$h(F,V) = \underbrace{4\cdot 3\cdot 4\cdot F}_{\text{halfedges}} + \underbrace{4\cdot 3\cdot V}_{\text{positions}} = 48F + 12V,$$

$$e(F,V) = \underbrace{4\cdot 4\cdot F}_{\text{indices}} + \underbrace{4\cdot 2\cdot F}_{\text{friends}} + \underbrace{4\cdot 2\cdot F}_{\text{sharpness}} + \underbrace{4\cdot 3\cdot V}_{\text{positions}} + \underbrace{4\cdot V}_{\text{loop}} = 32F + 16V.$$

Since $F \approx V$, Halfedge requires ca. 25% more memory than our method. Tab. 2 exemplifies this by measurements combining input and output buffer sizes.

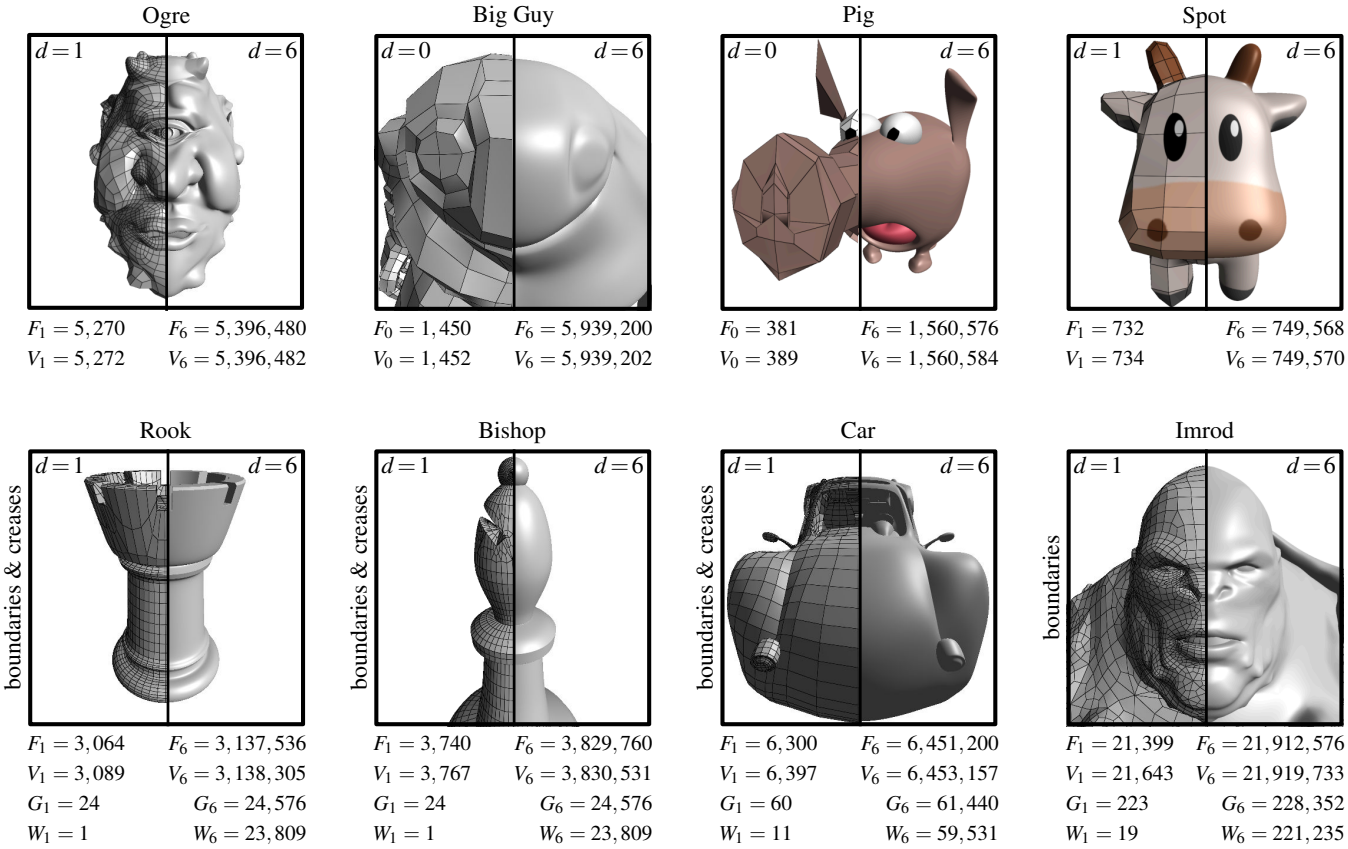

Figure 4: *Test Meshes. F_d denotes the number of faces and V_d the number of vertices. Meshes with boundaries require G_d ghost faces and W_d ghost vertices. Big Guy and Pig do not require a pre-processing iteration.*

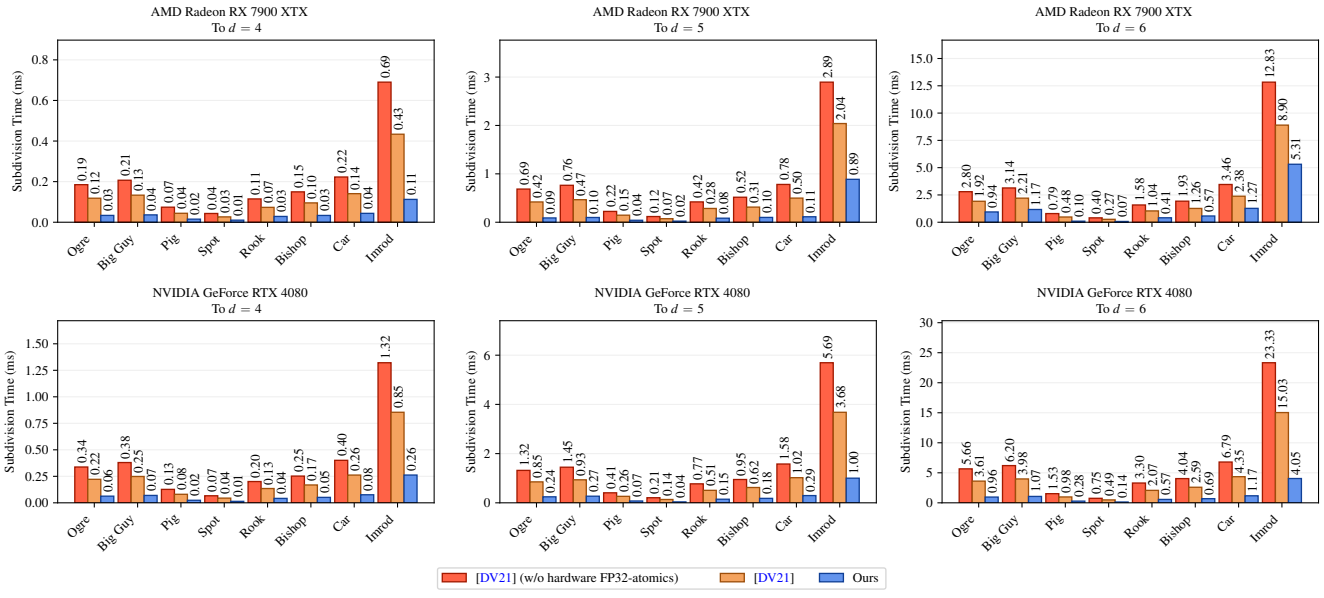

Figure 5: *Subdivision Performance. We compare our subdivision performance to level d against Halfedge [DV21] with and without hardware-supported atomic floating point arithmetic.*

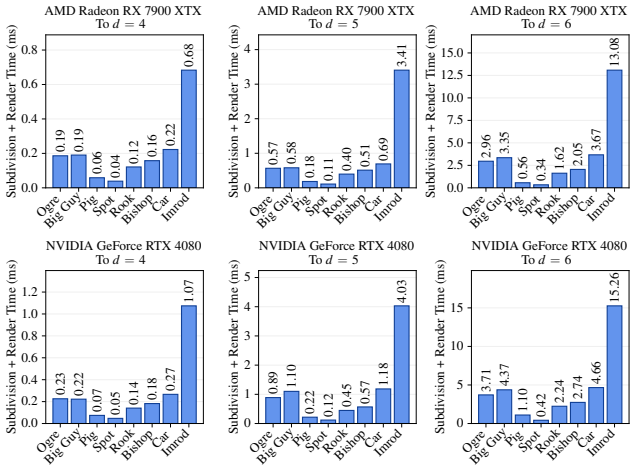

Figure 6: *Overall Performance. Our Direct3D12 implementation subdivides to level d and renders each mesh of Fig. 4 to a framebuffer of size* 1920×1080 *using the Blinn-Phong reflection model.*

Meshes	Ogre	Big Guy	Pig	Spot	Rook	Bishop	Car	Imrod
[DV21]	386.0	424.8	111.6	53.6	224.4	273.9	461.5	1567.4
Ours	308.8	339.8	89.3	42.9	180.9	220.5	372.7	1266.9

Table 2: *Temporary Memory Requirements for six subdivisions. Each cell is the temporary memory size in MiB required to subdivide the meshes from Fig. 4.*

As expected from a breadth-first subdivision method, both Halfedge and our algorithm are memory bound. To achieve a higher performance, the memory footprint has to be reduced. Our edge-friend achieves this by reducing neighbor information compared to Halfedge, as shown in Tab. 2. Additionally, by combining the edge- and face-point computation, we can make better use of loaded memory. We further benefit from an improved vertex memory layout, increasing the chances of cache hits. Moreover, we use regular writes into global memory which are faster than those with atomic additions. Unlike our approach, atomic-operations may cause flickering artifacts because of their non-deterministic scheduling. Finally, we only need a single synchronization barrier per iteration, while Halfedge needs three.

5. Conclusion and Future Work

We introduced a novel quad-based data structure for GPU parallel Catmull-Clark subdivision. Our algorithm is about three times faster compared to the latest related method. In future work, we want to expand our method to support adaptive subdivision based on surface flatness and distance to camera.

Acknowledgments

We thank Marc Stamminger, Dominik Baumeister, Carsten Faber, Holger Haupt, Pirmin Pfeifer and the reviewers. Meshes are courtesy of Keenan Crane (Ogre, Pig, Spot), Bay Raitt (Big Guy), Dmitry Parkin (Imrod), and OpenSubdiv (Rook, Bishop, Car). Open Access funding enabled and organized by Projekt DEAL.

References

[BFK*16] BRAINERD W., FOLEY T., KRAEMER M., MORETON H., NIESSNER M.: Efficient GPU rendering of subdivision surfaces using adaptive quadtrees. *ACM Transactions on Graphics 35*, 4 (July 2016), 1–12. 2

[BS02] BOLZ J., SCHRÖDER P.: Rapid evaluation of Catmull-Clark subdivision surfaces. In *Proceedings of the seventh international conference on 3D Web technology* (2002), pp. 11–17. 2

[CC78] CATMULL E., CLARK J.: Recursively generated B-Spline surfaces on arbitrary topological meshes. *Computer-aided design 10*, 6 (1978), 350–355. 2

[Cha74] CHAIKIN G. M.: An algorithm for high-speed curve generation. *Computer graphics and image processing 3*, 4 (1974), 346–349. 5

[dGDMD16] DE GOES F., DESBRUN M., MEYER M., DEROSE T.: Subdivision exterior calculus for geometry processing. *ACM Transactions on Graphics 35*, 4 (July 2016), 1–11. 4

[DKT98] DEROSE T., KASS M., TRUONG T.: Subdivision surfaces in character animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH '98* (1998), pp. 85–94. 2, 5

[DSSC08] DANIELS J., SILVA C. T., SHEPHERD J., COHEN E.: Quadrilateral mesh simplification. *ACM transactions on graphics (TOG) 27*, 5 (2008), 1–9. 6

[DV21] DUPUY J., VANHOEY K.: A halfedge refinement rule for parallel Catmull-Clark subdivision. *Computer Graphics Forum 40*, 8 (Dec. 2021), 57–70. 2, 4, 6, 7, 8

[HKD93] HALSTEAD M., KASS M., DEROSE T.: Efficient, fair interpolation using Catmull-Clark surfaces. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993), pp. 35–44. 6

[LB07] LACEWELL D., BURLEY B.: Exact evaluation of Catmull-Clark subdivision surfaces near B-Spline boundaries. *Journal of Graphics Tools 12*, 3 (Jan. 2007), 7–15. 2

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics 27*, 1 (2008), 1–11. 2

[MWS*20] MLAKAR D., WINTER M., STADLBAUER P., SEIDEL H., STEINBERGER M., ZAYER R.: Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the GPU. *Computer Graphics Forum 39*, 2 (May 2020), 335–349. 2

[Nas87] NASRI A. H.: Polyhedral subdivision methods for free-form surfaces. *ACM Transactions on Graphics 6*, 1 (1987), 29–73. 2

[NLG12] NIESSNER M., LOOP C. T., GREINER G.: Efficient evaluation of semi-smooth creases in Catmull-Clark subdivision surfaces. In *Eurographics (Short Papers)* (2012), pp. 41–44. 2

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics 31*, 1 (2012), 1–11. 2

[PEO09] PATNEY A., EBEIDA M. S., OWENS J. D.: Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of the conference on high performance graphics 2009* (2009), pp. 99–108. 2, 6

[Pix22] PIXAR RESEARCH: OpenSubdiv, 2022. URL: https://www.opensubdiv.org/. 2, 6

[RSS03] ROSSIGNAC J., SAFONOVA A., SZYMCZAK A.: Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces. In *Hierarchical and geometrical methods in scientific visualization* (2003), pp. 41–50. 3

[Sta98] STAM J.: Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th annual conference on computer graphics and interactive techniques* (1998), pp. 395–404. 2